AD-A173 049

(10)

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT unlimited This document has been approved for public release and sale; its distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION SPAWAR
6c. ADDRESS (City, State, and ZIP Code) Berkeley, California 94720	7b. ADDRESS (City, State, and ZIP Code) Space and Naval Warfare Systems Command Washington, DC 20363-5100
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

* Design Considerations for a Prolog Silicon Compiler

12. PERSONAL AUTHOR(S)

* Patrick C. McGeer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
technical	FROM _____ TO _____	* September 18, 1986	* 25

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Enclosed in paper.

DTIC ELECTE S OCT 14 1986 D E

DTIC FILE COPY

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Productivity Engineering in the UNIX† Environment

Design Considerations for a Prolog Silicon Compiler

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

Design Considerations for a Prolog Silicon Compiler

Patrick C. McGeer
(principal author)

Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720
(415) 642-4694

William R. Bush

Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720
(415) 642-4694

Jonathan D. Pincus

Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720
(415) 643-8229

Alvin M. Despain

Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720
(415) 642-5616

Design Considerations for a Prolog Silicon Compiler

ABSTRACT

We are designing a specialized silicon compiler using the programming language Prolog. This paper describes our design approach and our experiences in the initial phase of the project. Our compiler, ASP, is specialized for designing high-performance microprocessors. It is structured as a set of cooperating programs, written in Prolog, which communicate through constraint passing. The constraints are captured as notations on the system's single data structure. The design philosophy underlying ASP, its interface, major component programs and basic data structure are described. The choice of the language Prolog as the development and user language of ASP is discussed with some of its advantages and disadvantages illustrated.

Category number: 4

1. Introduction

The ASP (Advanced Silicon Compiler in Prolog) project's primary goal is the production of a silicon compiler that can create, from a high-level specification, a very high performance processor with minimal designer interaction. It happens that we are not only interested in using Prolog as a design language, but also in designing high-performance microprocessors specialized for the execution of Prolog[Dob85]. As a result, we are investigating new techniques in such areas of automated chip design as timing-directed global routing, optimization of pipelined bus organization and control, and an automatically generated, parameterized off-chip interface.

Simultaneous progress on many fronts requires, as a first step, the development of an environment in which experiments may be performed and the tools can be integrated. The ASP environment includes an array of integrated simulation and debugging tools, as well as a usable skeleton of the overall compiler. Given the skeleton, each module of the compiler may be improved independently.

2. Silicon compilers

David Johansen of Caltech first proposed the idea of a silicon compiler in 1979 [Joh79]. Much of IC design is a fairly mechanical task of putting blocks into prearranged slots and routing between them; a silicon compiler simply automates this job. Silicon compilation is distinguished from design tools -- such as layout editors and stick compilers -- by the amount of designer interaction required and the level of abstraction dealt with by the tool.

Layout editors such as Magic, Caesar and Kic-2 require the designer to lay down paint. Although Magic permits the designer to view cells as physical abstractions at higher levels through use of a cell hierarchy, there is no automatic translation between different levels of abstraction: a user cannot enter a circuit at an arbitrary level of abstraction and expect the tool to generate the paint layers for him. Thus interaction is continually required at every level of the design.

LAVA and Sticks-style compilers allow the user to specify the circuit in terms of sticks as opposed to paint. This is a slightly higher level of abstraction, and translation between the two levels is automatic, but the description is still on a relatively low level.

In a silicon compiler, however, the designer inputs a description of the chip at some suitably high level of abstraction, and the program performs the translation down to the physical level. The designer should not be called on to aid the program in its transformation; any design decisions should be specified with the description.

The analogy with programming language compilation is tempting. Layout editors are equivalent to being able to input an program as a file rather than having to toggle it in through the front panel. A LAVA-type program is more like a symbolic assembler: some additional abstraction is permitted, but the correspondence to the object code is still extremely close: in order for translation to be called compilation, the language being translated must be relatively high level. Finally, when compiling a program, it is certainly not desirable for the

programmer to be continually forced to guide the compiler down the correct paths.

3. Previous work

In addition to defining the term, Johannsen also designed *Bristle Blocks*, [Joh79] generally heralded as the first silicon compiler. It uses parameterized cells; his initial program only designed the data path, but along with a channel router and a control generator, it is now being marketed as Genesil by Silicon Compilers, Inc. *Bristle Blocks* is a relatively low level compiler; the designer views the circuit as a combination of the pre-defined cells.

Lincoln Labs' *MacPitts* [SiSoC82] took a high-level description of the chip in a LISP-based HDL. The original version used fixed cells along the data path, a channel router based on the Rivest-Fidducia approach, and a single Weinberger array for control. The current version, marketed by MetaLogic, Inc., as MetaSyn, includes parameterized cells and a more sophisticated partitioned control, consisting of a combination of PLAs and Weinberger Arrays.

4. Language issues

Most CAD systems are written in C or LISP; the choice of Prolog may appear unusual. Several features of Prolog, however, make it particularly well-suited for our purposes.

4.1. Prolog As A Specification Language

Prolog differs from most other programming languages in that its semantics are tuned towards the description of a problem, rather than the implementation of its solution. This implies that the distinction between program and data is much less clear than it is in a conventional programming language. In hardware terms, the language is inherently a *specification* language; no constructs or operators are required other than those provided in the language itself.

4.2. Isotropy

An important feature of any design tool is isotropy: the language and environment that a user or programmer sees should be independent of either his position in the design space or his level of sophistication. This quality is rarely espoused, but it is extremely important. DA systems written in Lisp have proven effective since a user has learned Lisp in the course of learning the application-specific Lisp subset that he acquires when he learns the DA system. This permits such a user to extend and tailor the system to meet his needs.

4.3. Automatic Parallelism Detection

Parallelism detection is much more important in hardware design than in software design. Most programs run on uniprocessor hardware, and so most optimizations other than constant folding, loop unwinding and the elimination of redundant code have little effect. However, hardware designers face no such restrictions: indeed, one of the principle arts of the microarchitect is the parallel

utilization of hardware.

Our research group's efforts at parallelism detection for multiprocessing Prolog applications [Cha85] are, we believe, directly applicable to hardware design. The key to this is that much of the potential parallelism of a Prolog program, as opposed to a Lisp or C program, is inherent and explicit in the code. It seems clear that the parallelism of a Prolog hardware specification should also be immediately apparent.

4.4. Prolog's Built-In Database

A silicon compiler requires a large database component, if for no other reason than the sheer amount of information present. Prolog contains a built-in relational database: indeed, one can argue that Prolog is merely a generalization of a relational database. Hence this aspect of the compiler is already completed.

4.5. Support for Rule-Based Systems

Since no polynomial-time algorithms exist for many of the tasks involved in silicon compilation, it is likely that individual modules of ASP may best be structured as expert systems. Hence we desired a language which permits rule-based programming without major modification. Prolog obviously meets this requirement.

4.6. The Logical Variable

Finally, Prolog permits a form of lazy evaluation. It is possible in Prolog to equate numerous variables in a user-transparent fashion through the Prolog

mechanism of unification. When this is done, assignment of a value to any one sets all the values. These assignments are all conditional; they are automatically undone if the particular heuristic at which they were bound fails.

4.7. Other Considerations

It has become part of the programmers' folklore that Prolog is too slow for production systems. This may be because of the almost universal use of Prolog interpreters, which run at about 1500 LIPS (Logical Inferences/second) on a Vax 11/750. However, we have constructed a high performance processor and compiler for Prolog, that runs at about 300 kilolips, or roughly two orders of magnitude faster than most systems used by Prolog programmers[Dob85].

5. General design of ASP

ASP is conceived of as a number of cooperating programs hung on a skeleton, rather than a single large program. This organization produces a number of benefits. First, it facilitates an incremental development strategy: programs may improve so long as they do not change their interfaces. A second benefit is the trail of tools left behind for those wishing to use some of the algorithms and methods of ASP without using the entire package. Finally, this approach permits human intervention in the place of any particular program; this is particularly useful in the development stage, when individual components may not yet be completed. Of currently-reported silicon compilers, Bell Labs' CADRE [Ack85] comes closest to this overall organization.

From a software engineering standpoint, such a design can work only if the organization of the overall package is simple. Furthermore, since individual components may or may not be present, interfaces must be identical (or nearly so). There are three reasons for this. First, one program should not be dependent upon the presence of any other program within the overall environment; its only dependency should be upon the presence of data. Second, no program should specify or use more than a single output or input interface; it is clear that if each program specified its own input interface, then each program would have to specify n output interfaces. Worse, if any new programs were added to the ASP system, the programmer would be forced to modify the output routines of each other program in the system. Third, no area in design automation is mature: programs and techniques are evolving rapidly. Hence we wish to provide for independent evolution of each component. This in turn requires that the evolution of any single component be transparent to the remainder of the system.

Ideally, then, all the programs should use the same data structure. Although this simplifies the interface problem, it puts a great deal of stress on the structure itself: it must be easily described, accessed, and modified, yet flexible enough to accommodate a wide variety of applications without modification.

6. The Constrained Hierarchical Schematic

ASP's basic data structure is the *constrained hierarchical schematic* (CHS). A CHS consists of basic circuit elements and other CHS's which serve to

describe the circuit in a hierarchical fashion. Each CHS is constrained by a series of notations, which serve as a full description of the electrical, structural, and behavioral properties of the cell in question. These notations include size constraints on the bounding box, protection frames for each mask layer, the presence and coordinates of feedthroughs, the name of the circuit element, constraints on its performance, on the capacitive load of each input and drive resistance of each output.

6.1. CHS Definition

A Constrained Hierarchical Schematic (CHS) is a data structure realized as a Prolog clause, with the following fields (coordinates are always non-negative integers in terms of a single parameter, lambda):

(1) Name -- the name of the function that this CHS implements.

(2) Bounding Box -- a height/width pair giving box dimensions.

(3) Ports -- a list of physical connections, data structures of the following form: port(Signal, X1, Y1, X2, Y2, Levels) where Signal is the name of the signal carried at the port, and X1, Y1, X2, and Y2 are the x,y coordinates of the lower left and upper right coordinates of the port, and Levels is a list of the Levels on which the signal may be carried.

(4) Inputs -- a list of data structures of the form:

signal(SigName, DriveBy, Capacitance, Resistance)

where SigName is the name of the signal, Capacitance is the capacitance (in pf) of the load, Resistance is the total resistance of the load in ohms,

and DriveBy is the time to charge or discharge the node in nanoseconds.

(5) Outputs – a list of data structures of the same form as input; in this case, the variables do not *specify* capacitance, resistance, or required drive time; they state capabilities.

(6) Substructure – a list of *instances* of CHS's, wires, transistors and vias which make up the CHS.

(7) UC_In – a list of the *unbound constraints* that must not be violated by this CHS (or, if you prefer, a list of the unbound constraints of the system when this CHS is instantiated);

(8) UnboundConstraints – a list of the unbound constraints of the system following the instantiation of this CHS.

Every field here should be self-evident, save the UC_In and Unbound Constraint fields. The purpose of these will be explained below.

The instance of a CHS is the CHS with two additional fields: Position, which is the position of the CHS within its parent; and Orientation, which is one of the eight Manhattan transforms within the plane.

6.2. CHS Constraints And the Body of A CHS

A consistent problem in automatic IC design has been the expression of constrained variables in a design template. For example, consider the static CMOS inverter depicted in figure 1. Both the input port and the output port are constrained in that they must lie on the left and right boundaries, respectively, and must lie between the rows reserved for the p and n transistors.

Further, the ultimate values of the rows used for the ports must be bound into the substructure, since these values determine the position of the wires that run between the ports and the connection to their signal columns.

We found an elegant solution in the logical variable and the natural representation of both code and data in Prolog. In Prolog, a procedure is a sequence of clauses; and each clause is of the form:

clausehead :- goal1, goal2,..., goaln.

which is read "clausehead is true iff goal1,...,goaln are true".

This representation led us to the idea that a CHS is best represented as a clause with the head as the structure described above, with unbound variables for any quantities unspecified, and the body as a set of constraints on those quantities.

This can be made clearer with an example. In figure 2, we give the relevant subset of the inverter CHS, and its body. Ellipses mark elided arguments and irrelevant aspects of substructure.

The pictured here, with the pictured subset of its associated constraints, specifies an inverter whose power and ground and input and output ports are constrained to lie on the left and right edges of the inverter's bounding box, but may lie anywhere on those edges subject to the constraints given. The constraints given are derived from the technology design rules, and are the subgoals of the inverter's CHS clause. The constraint names should be read as "ge" for ">=", "eq" for "equal", and so forth.

```
chs(  inverter,
      box(Xsize, Ysize),
      [     port(vdd,0,0, VddLine, VddLine1, [metal1]),
            port(vdd, 0, VddLine, Xsize, VddLine1, [metal1]),
            port(gnd, 0, GndLine1, XSize, GndLine, [metal1]),
            port(gnd, 0, GndLine1, Xsize, GndLine, [metal1]),
            port(A, 0, Yin1, 0, Yin, [metal1]),
            port(B, XSize, Xsize, Yout1, Yout, [metal1])],
      [signal(A,...,...,...)],
      [signal(B,...,...,...)],
      [...,wire(vdd, 0, VddLine, Xsize, VddLine1, metal1),
            wire(gnd, 0, GndLine1, Xsize, GndLine, metal1),
            wire(A, 0, Yin1, X3, Yin, metal1),
            wire(B, X4, Yout1, XSize, Yout, metal1),
            contact(A, X3, Yin1, X5, Yin2, metal1, poly),
            ...],
      UnboundConstraints) :-
      eq(VddLine1, VddLine + 2, [], UC1),
      eq(GndLine, GndLine1 + 2, UC1, UC2),
      le(VddLine1, Ysize, UC3, UC4),
      ge(X3, 0, UC4, UC5),
      eq(X5, X3 + 5, UC5, UC6),
      eq(Yin, 2 + Yin1, UC6, UC7),
      eq(Yout, 2 + Yout1, UC7, UC8),
      ge(VddLine, Yin + 3, UC8, UC9),
      ...,
      ge(Yout, GndLine + 3, UCn, UnboundConstraints).
```

Figure 2 - Partial CHS Description of Inverter, with Relevant Constraints

This CHS returns (or, in Prolog parlance, "succeeds") iff all the subgoals are satisfied. Further, the unification properties of the logical variable assures that every instance of any logical variable is bound to the same value. Hence, we are assured that any inverter which meets the constraints above will be returned by this clause.

A question that occurs immediately is the following: what happens if the variables are unbound when the clause is entered? This is the function of the UC_In and Unbound Constraint fields in the CHS clause, and the third and fourth fields of each subgoal. Constraints which must be met but cannot be checked, since they involve one or more unbound variables, are placed on the list of unbound constraints. The maintenance of this data structure and its resolution are detailed below; for now, we note that this variable permits this specification clause to become a generation clause.

In addition, it is possible in Prolog to define several clauses for one procedure; hence, several different inverter specifications may be given, and any attempt to generate an inverter will try each in turn until one succeeds or all have failed.

6.3. Substructure and the Body of a CHS

When a CHS contains other CHS's as part of its substructure, the body of the CHS clause acts as a generator of the sub-CHS's. It does this in two ways: first, it instantiates the CHS's through a provided routine instantiate_chs; second, it acts to propagate constraints between the child CHS's. In particular, it acts to pitch-match the ports on the sub-CHS's.

Consider, for example, the adder bitslice presented in figure 3. This adder, based on the Manchester Carry Chain, is composed of three parts: logic to generate the propagate/kill/generate signals; carry chain logic; and sum generation logic.

We wish to consider the example of an interface between CHS's, and for this purpose we'll look at the interface between the CHS's representing the carry chain and the propagation logic block. The relevant code appears in figure 6.

In this clause, instantiate_chs is a routine which looks for CHS's of the name of its first argument, which can fit into the box defined by the second and third arguments if the CHS is oriented as specified by the fourth argument and if the unbound constraints given in the fifth argument are met and modified to the unbound constraints given in the sixth argument; the seventh argument contains the template of the CHS which has been instantiated. findPorts is a

```
chs(adder,
      box( XSize, YSize ),
      ...,
      [...,
            chs_struct(propLogic,position(X1,0),position(X2,Ysize),Orient1,UC1,PL),
            chs_struct(carryLogic,position(X2,0),position(X3,Ysize),Orient2,UC2,CL),
            ...],
      UC_In,
      UnboundConstraints) :-
      instantiate_chs(propLogic,position(X1,0),position(X2,Ysize),Orient1,UC_In,UC1,PL
      instantiate_chs(carryLogic,position(X2,0),position(X3,Ysize),Orient2,UC1,UC2,CL,
      findPorts(PL, position(X1,0), Orient1, x, X2, Ports1),
      findPorts(CL, position(X2,0), Orient2, x, X2, Ports2),
      matchPorts( Ports1, Ports2, UC2, UC3 ),...
      ...;
```

Figure 6 - Body of A Clause which Contains CHS's.

routine which finds all ports on the given CHS, if its lower left corner is as specified by position and its orientation is specified by orientation. It finds all ports on the boundary marked by its fifth argument for the dimension given by its fourth. matchPorts reconciles two lists of ports, if possible, again updating the set of unbound constraints.

6.4. Unbound Constraints

The above works quite well if all the logical variables are bound or instantiated when the CHS is instantiated. Often, however, this is not the case: in particular, the location of an output port may be bound to an unbound variable representing a location in the parent cell. In this case, the constraints are attached to the unbound constraint field in the instantiation of the CHS, and whenever a variable in the instantiation of the CHS is bound to a value, the unbound constraint field is checked for constraint violation, and the constraint list is updated.

The constraints on a variable are kept as follows. If a variable is unbound but constrained, it is set to a data structure of four values: the value of the variable (currently unbound); upper and lower bounds on the variables value (initially set to infinity and -infinity) and a list of constraints on the variable. When a new constraint is added, the bounds are updated, if possible, and the constraint list is checked; if the new constraint is incompatible with other constraints, the chs clause fails; if the new constraint is redundant, nothing is changed; otherwise, the new constraint is simply added, and constraints on variables which are constrained by this variable are updated, recursively.

These lists are maintained by the constrained routines eq, ge, le, etc.

At the end of the design process, the unbounded constraint variable may be empty, in which case the design is completely specified. If it is not, however, then the final specification of the design involves an integer programming problem. However, we believe that this integer programming problem will be relatively easy to solve, since *any* feasible solution is acceptable and since we believe that in practice most designs are not underconstrained.

6.5. Implications of the CHS

This schematic clearly permits hierarchical structures to be modeled. Furthermore, it allows different programs to view the circuit at various levels of abstraction. For example, the chip floor planner divides the circuit into two CHS corresponding to the control and data path respectively, not worrying about any greater detail, while the layout generator deals with the mask level.

One great advantage of the *isotropy* of the CHS – the fact that a CHS can contain CHS as part of its description – is the possibility of algorithms which work on multiple levels of abstraction. For example, since both individual gates and bitslice cells within the data path (such as the ALU, shifter, etc.), are represented as CHS, both our timing analyzer and our layout generator, which were designed to work at the gate level, are able to work at the data path cell level. We expect this property to aid the development of multi-level simulation.

The choice of the CHS as the principal data structure is related to the choice of constraint propagation as the design methodology. A good design is one that meets constraints. At the user level, a chip is merely a piece of silicon

that must be less than a given area, performs a given function in a given time and consumes less than a given amount of power. ASP works by partitioning the constraints at any level among component pieces, and then designing the pieces.

7. ASP Chip Methodology

In addition to a design methodology and a central data structure, a silicon compiler requires a chip methodology: a partitioning of the chip generation software into modules. In this section we briefly present our chip methodology and relate it to the constrained hierarchical schematic.

Our divisions are based on Pendleton's [Pen85]. We identify seven partitions in chip design: register transfer, interface, data path unit, data path organization, control unit, control organization, and interconnect. Above these is general microarchitecture design. The task at this higher level of design is the translation of an ISP-like description to some register-transfer description of the chip. We believe the major performance gains and interesting research issues tend to lie here, but we are currently devoting our efforts to the lower levels, since they are better understood, and since they must be done reasonably well before we can hope to successfully address the major issues at the microarchitecture level. The lower levels of design form a substrate upon which we hope to build a microarchitectural expert.

7.1. Register Transfer

The task at this level is to map the behavioral RTL description produced at the microarchitecture level, by tools or a human architect, into CHS circuit structures. The RTL description mechanism is tuned to high performance processors rather than algorithmic style. The translator will produce both functional simulations for design verification and CHS's.

7.2. Interface

ASP's strategy for this level is to use design frame technology [Bor85]. For a given bus, a design frame and a universal interface providing pads, control PLA, and data registers will be supplied by ASP.

7.3. Data path Unit

There are at least four approaches to the problem of functional unit generation: standard cells, parameterized cells, module experts and module generation. The organization of ASP as a set of cooperating programs permits the use of any such approach. The program which translates the data path cell description to a physical layout can either make use of a standard cell library or generate the layout itself; it makes no difference to the other programs comprising ASP.

7.4. Data path Organization

The problem at this level is, given a bus/block-connection schematic, arranging the data path blocks so that constraints are met on the total length of any bus and on the total width of the widest cell in the data path. A variety

of approaches, from force-directed placement through variations on Wing's algorithm through breadth-first search heuristics are in experimental use here. The interesting thing to note here is that this is the same problem as block-placement in non-folded CMOS gate matrices, and can be attacked by the same algorithm, a non-obvious identity that becomes clear when it is realized that a CHS may represent either a data path or functional blocks within a gate matrix.

7.5. Control Unit

As with data path unit generation, there are a number approaches to the problem. Control units may be implemented as ROMs, PLAs, Weinberger arrays or gate matrices. Similarly, the control unit generator can choose a physical layout independently from other ASP tools.

7.6. Control Organization

Control partitioning, state encoding, and microorder decoding are dealt with at this level. One area under active exploration is functional partitioning of control structure and functional state encoding. An example of the latter may be found in Ullman[Ull84]: the justification is that for most forms of logic layout (except for ROMs) the number of state bits is not a metric that should be optimized.

7.7. Interconnect

In microprocessor design, routing may be said to have three major components: global routing (from the pads to the interface registers), maze

routing in the data and control paths, and routing in the control path/data path channel. Global routing is fixed as part of the interface design. In the data path, horizontal routing is done in first layer metal, area provided for by either feedthroughs in the cells (for parameterized or generated cells) or by an routing channel over each bitslice (for fixed cells), and vertical routing is done in second-layer metal and hence is transparent to horizontal routing. Timing-directed routing is being explored for inter-path routing, since the timing constraints are easily captured by the CHS.

8. Other integrated tools

In principle, each of the levels of chip design can be viewed as a separate tool. Each should be able to stand on its own, but they are designed to work cooperatively. A few other important tools do not fit so nicely into the decomposition above; in general, these will be used as components in more than one of the levels.

8.1. Multi-level simulation

The phrase *multi-level* simulation refers to the possibility of simulation occurring at different levels of abstraction. A circuit simulator such as SPICE views the chip at an extremely low level, while a switch-level simulator like ESIM works at a higher level, and a gate-level simulator still higher. This corresponds to different elements – wires, transistors, or gates, respectively – of the circuit being viewed as primitives.

Two kinds of simulation are required: correctness checking and timing analysis. For example, an RTL checker can be used to test whether there is a difficulty at this level; if so, there is no point in continuing the translation process. Multi-level simulation is facilitated by the isotropy of the data structure. Similar algorithms should be used to perform the simulations on different levels. At the very lowest level, however, ASP will simply provide an interface to SPICE.

8.2. Transistor Sizing and Ordering

The control and module generators will generate functionally correct transistor-level schematics. In order to achieve desired performance criteria, however, it may be necessary to attach size constraints to various transistors to speed up the circuits critical paths. Furthermore, in many cases the order of series transistors can not be determined until some timing information is known. The MOST (Method for Ordering and Sizing Transistors) program [PiDe86] takes care of both of these tasks.

Clearly these tasks require some level of timing analysis. However, since MOST occurs before layout, the estimations of interconnect resistances and capacitances will necessarily be only approximations. Thus, the timing analyzer is on the level of Crystal, rather than Spice; this is also important for speed reasons.

9. Conclusion

ASP is both a silicon compiler and a vehicle for research and experimentation in automated design of CMOS microprocessors. The framework of ASP is therefore designed to permit incremental improvement and evolution. This is accomplished in three ways: the use of a single flexible data structure as a common interface to all component programs, constraint-passing as the sole mode of communication between programs and the choice of an implementation language that is an extension of a relational database, which permits the natural modeling of the data structure and the natural inclusion of rule-based, database and algorithmic methods.

10. Acknowledgements

The authors wish to acknowledge the many fruitful and enlightening conversations we've had with our colleagues at UC-Berkeley. In particular, Walter Scott, Bob Mayo, Gordon Hamachi, Joan Pendleton, Gaetano Borriello, Fred Obermeier, Randy Katz and Glenn Adams have been most helpful. Andrew Kahng read early drafts of this report and provided many helpful suggestions. Our colleagues on the Aquarius Project at UC-Berkeley have provided much moral and intellectual support. This research was sponsored by the Defense Advanced Research Projects Agency under Order 4871, monitored by the Naval Electronic Systems Command, Contract N00039-84-c-0089.

11. References

[Ack85] Ackland, B., et al, "CADRE -- A System of Cooperating VLSI experts",

[Bor85] Borriello, G., and Katz, R., "Design Frames: A New System Integration Methodology", *1985 Chapel Hill Conference on VLSI*, May 1985.

[Cha85] Chang, J. H., "High Performance Execution of Prolog Programs Based on Static Data Dependency Analysis", *Ph D Thesis*, Computer Science Division, University of California at Berkeley, Berkeley, CA, 94720.

[Dob85] Dobry, T. P., et al., "Performance Studies of a Prolog Machine Architecture", *Proceedings of the Computer Architecture Conference*, June 1985

[Joh79] Johannsen, D., "Bristle Blocks: A Silicon Compiler", *Proc. 16th Design Automation Conference*,, June 1979.

[Pen85] Pendleton, Joan M., "A Design Methodology for VLSI Processors", *Ph D Thesis*, Department of Electrical Engineering and Computer Science, UC Berkeley, Berkeley, CA, 94720, 1985.

[PiDe86] Pincus, J. and Despain, A., "Delay Reduction using Simulated Annealing", *these proceedings*, June 1986.

[SiSoC82] Siskind, J., Southard, J., Crouch, C., "Generating Custom High-Performance VLSI Designs from Succinct Algorithmic Descriptions", *Proc. Conf. Advanced Research in VLSI*, P. Penfield., Ed., MIT, Cambridge, MA, Jan 1982.

[Ull84] Ullman, J. D., "Computational Aspects of VLSI", Computer Science Press, 11 Taft Court, Rockville, MD, 20850

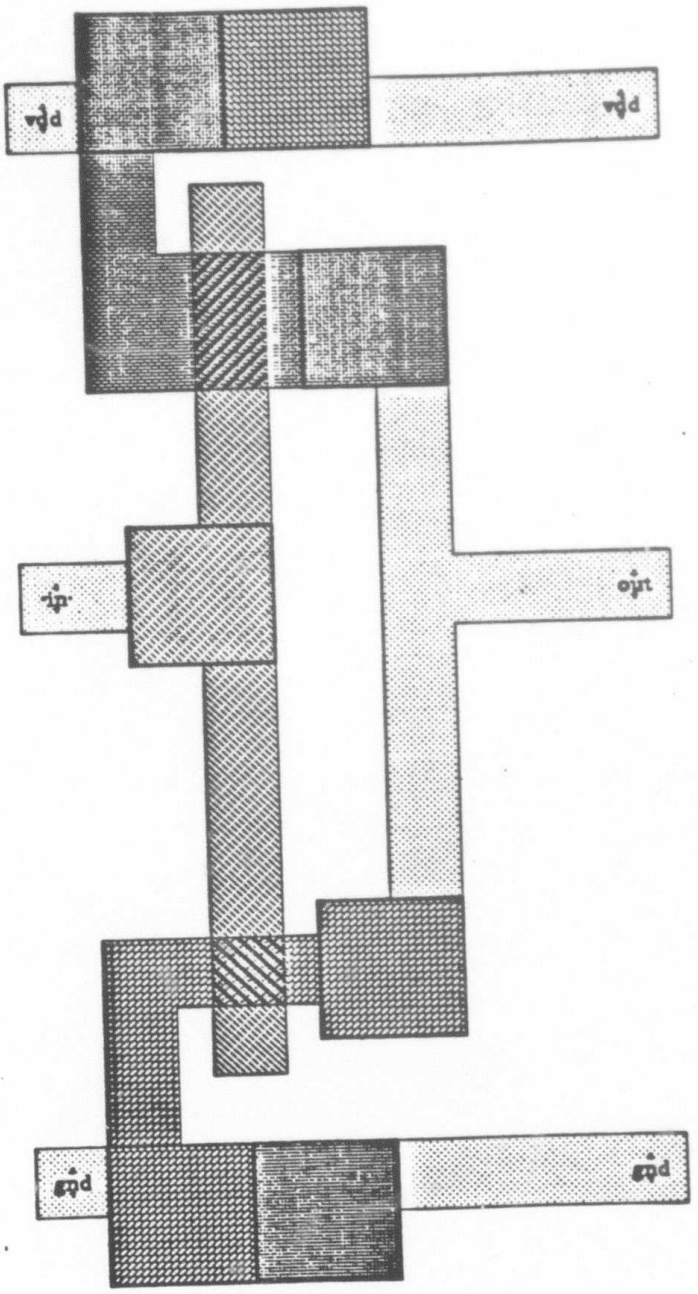

Figure 1: A Standard Inverter

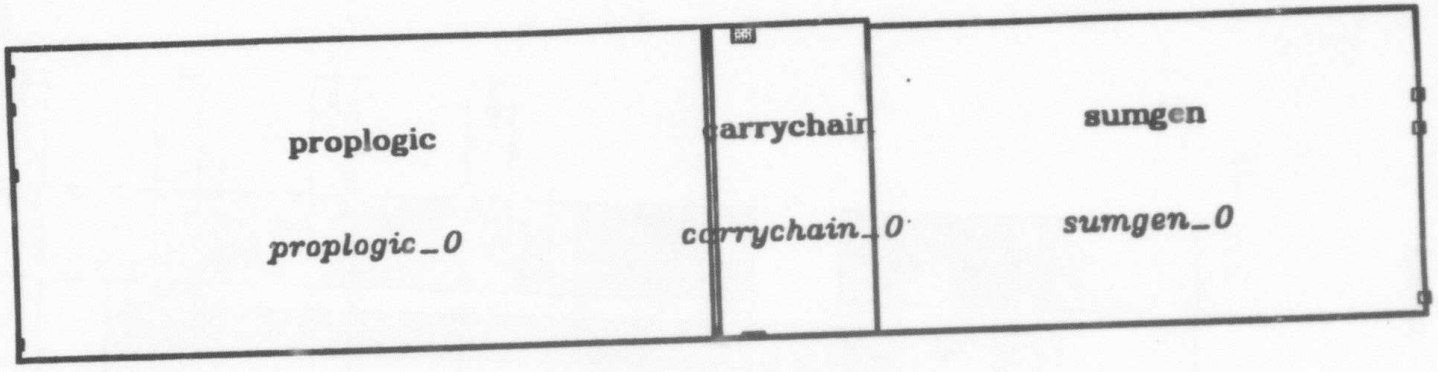

Figure 3: A Bit Slice of an Adder

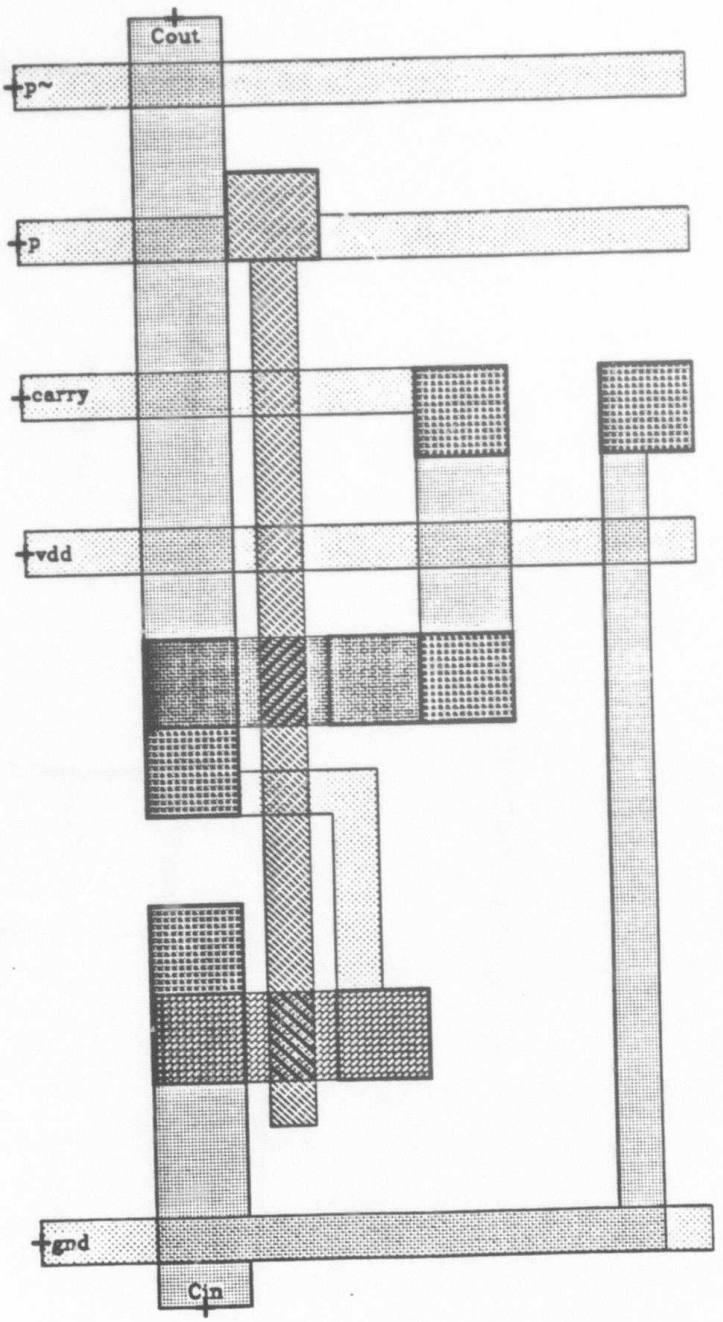

Figure 4: Carry Logic

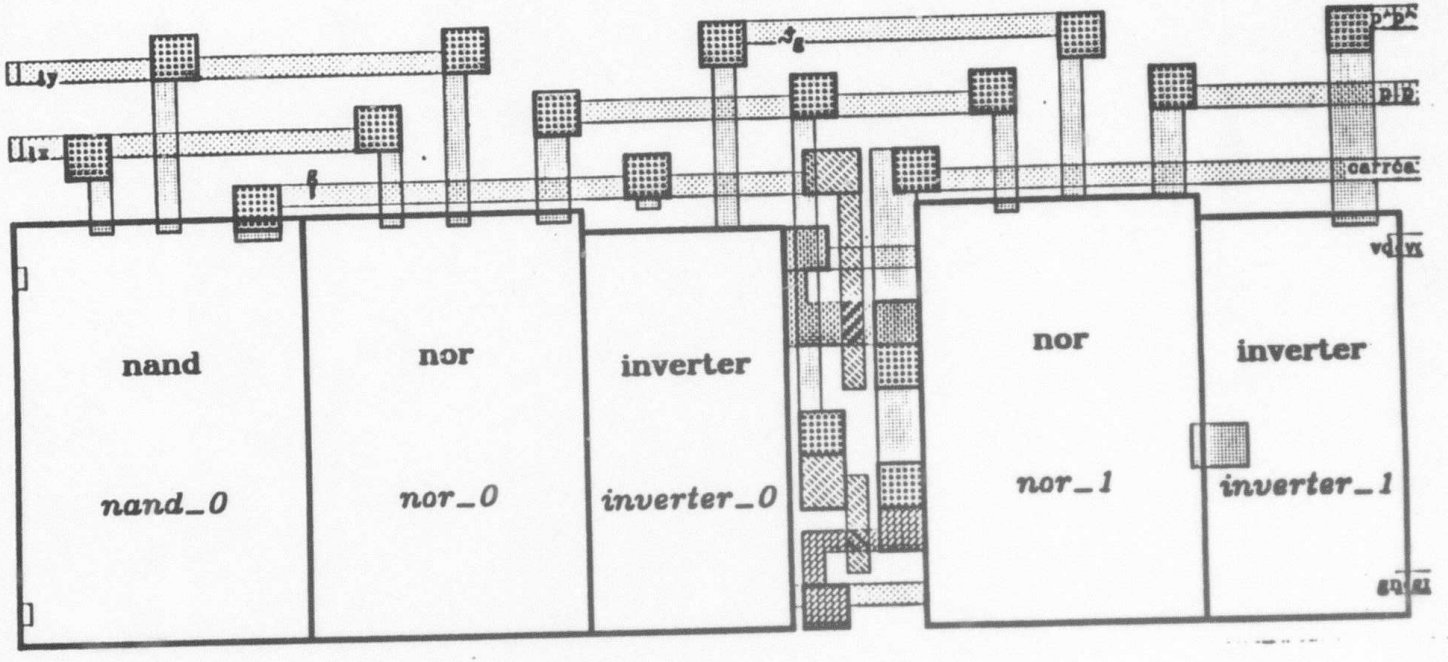

Figure 5: Propagate Logic